

Portfolio

Shin Beom Su

“상상을 현실로 만들 수 있는 개발자 신범수입니다.”



게임 클라이언트 개발자

신범수

게임은 상상을 현실로 만드는 강력한 도구입니다.

게임 개발은 이상을 바탕으로 현실을 설계하는 행위입니다.

눈에 보이지 않는 개념을 갖고 눈에 보이는 결과를 만들어야 하기에, 많은 실패와 장애물이 있습니다.

하지만 **실패에서 돌파구를 찾고 스스로 장애물을 넘는 방법을 만들 줄 알아야** 상상을 현실로 만들 수 있다고 생각합니다.

저는 인디 개발팀을 이끌며 많은 장애물을 만났고, 사람들과의 **협동**으로 이를 넘어왔습니다.

그렇게 쌓은 **3번의 정식 출시 경험**과 **4번의 수상 실적**은 앞으로 동료들과 맞설 또 다른 장애물 앞에서 쓸모 있는 이정표가 되어줄 것입니다.

Mail : qjatn147852@naver.com

Github : <https://github.com/shin0624>

Blog : <https://shin0624.tistory.com>

Unity

3ds Max

Git

Android

C#

C++

Java

HERESIS

#3D #탈출 #공포 #Steam 출시작

Unity

3ds Max

C#

Git

Steam

담당 업무

[클라이언트 개발]

● 에너미 AI 개발

- 행동 트리와 AI Navigation을 사용한 에너미 의사 결정 로직 구현
- Mixamo, Animator를 사용한 에너미 FSM 구현

● 유저 경험 향상 요소 개발

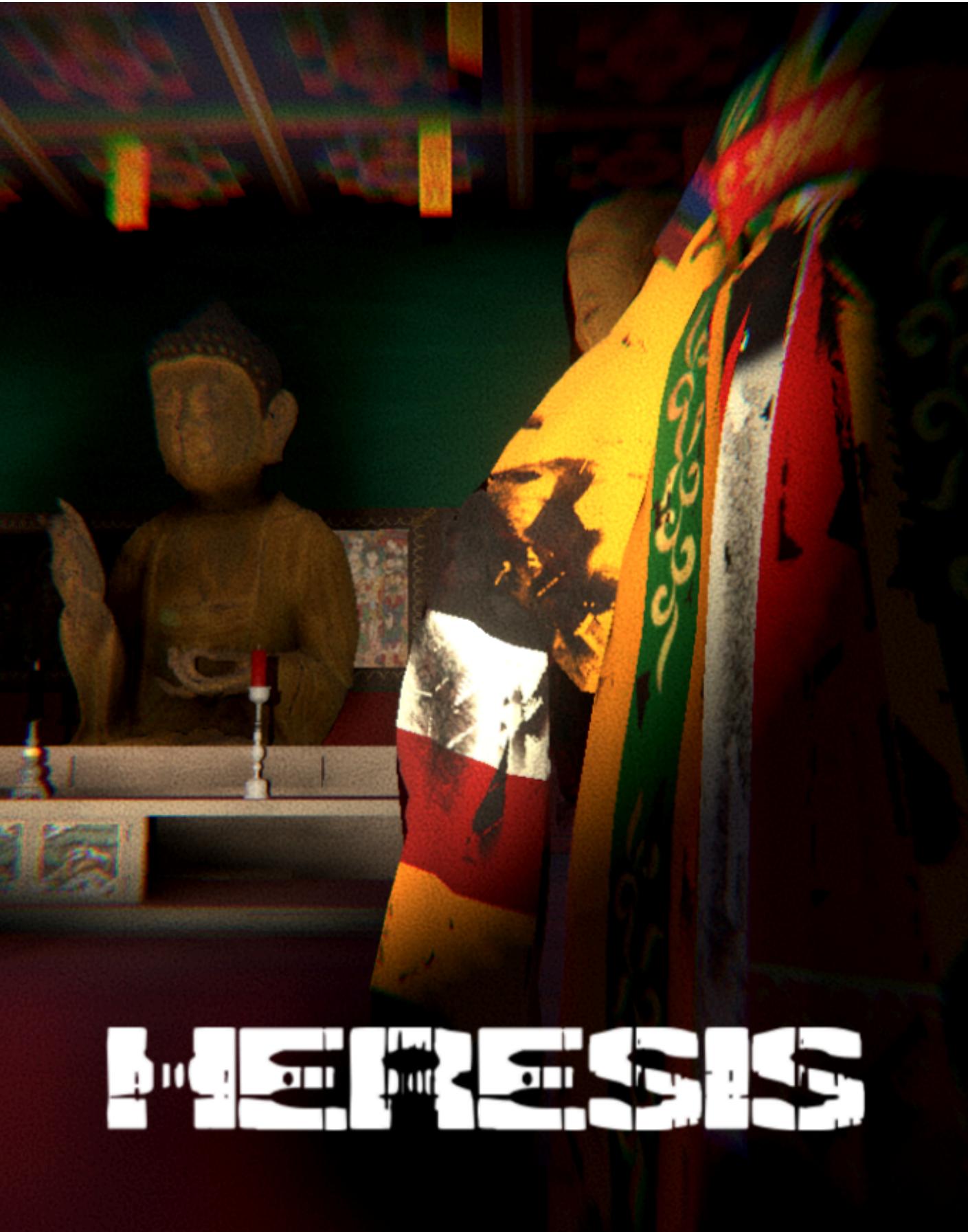
- Perlin Noise 알고리즘을 활용한 피격 시 카메라 진동 효과 구현
- 마우스 클릭으로 입력 받은 패스워드를 처리하는 도어락 UGUI 구현

● 시네마틱 컷신 개발

- Timeline, Cinemachine을 사용한 인트로, 이벤트 컷신 구현
- Dolly Camera와 Virtual Camera를 조합하여 자연스러운 공간 전환 효과 구현

[프로젝트 관리]

- SteamWorks 파트너 등록 및 Steam 출시 전담
- Git Repository 및 PR 관리, 사용법 공유
- 출시 후 피드백 기반 1차 업데이트 전담



[GitHub](#)



[Steam](#)

Behavior Tree와 AI Navigation을 사용한 에너미 AI 구현

보스 에너미의 의사 결정 알고리즘을 Behavior Tree 기반으로 구현하였습니다.

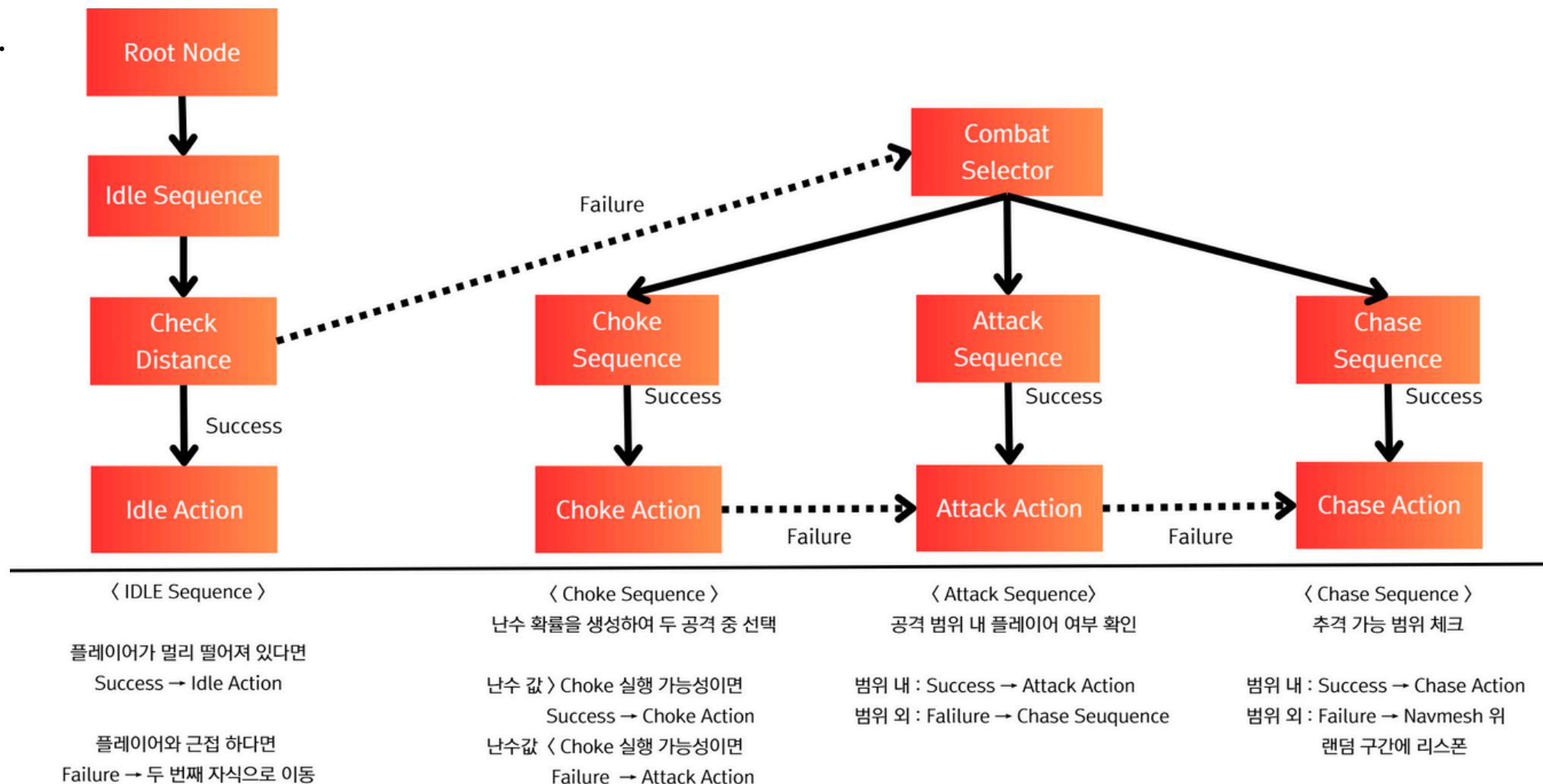
[의도]

- INode 인터페이스를 상속 받지 않고 추상 부모 클래스 기반 행동 트리를 구현하여, 트리의 구조와 동작을 이해하고자 하였습니다.
- 단순한 FSM보다 더 유연한 의사 결정이 가능한 AI를 설계하고자 하였습니다.

[동작 원리]

핵심 동작 노드

- BehaviorNode : 모든 노드의 부모 클래스. 추상 메서드로 선언
- SequenceNode : 모든 자식 노드를 순차적으로 실행.
- SelectorNode : 자식 노드 중 하나라도 성공하면 Success를, 모든 자식이 실패하면 Failure를 반환.



행동 노드

- IdleActionNode : 플레이어가 근접하면 공격 / 추격 상태로 전환.
- ChaseActionNode : 플레이어가 추격 가능 범위에 있을 경우 추격 수행.
- AttackActionNode : 공격 범위 내에 플레이어 존재 시 공격 수행.

[직면한 문제]

- 에너미가 플레이어를 발견해도 다음 시퀀스로 전환되지 않는 문제
- 플레이어 추격(Chase)에 실패한 후 IDLE 상태가 무한 반복되는 문제



문제 해결 과정

1

문제 상황 인식

에너미가 플레이어를 발견해도 다음 시퀀스로 전환되지 않음.

2

원인 파악

- 에너미는 플레이어와 근접하기 전까지 IDLE 상태가 지속됩니다.
- 이 때 **Running**이 리턴되었기 때문에 AI는 계속해서 “행동이 진행 중”이라고 판단하여, 플레이어가 근접해도 IDLE이 **무한히 실행 중인** 상태가 유지되어 버립니다.

```
- public class IdleActionNode : BehaviorNode
{
    ...
    //idle 애니메이션을 실행하는 노드
    private ShamanController shaman;
    public IdleActionNode(ShamanController shaman)
    {
        this.shaman= shaman;
    }
    public override Status Evaluate()
    {
        shaman.animator.SetTrigger("IDLE");
        return Status.Running;
    }
}
```

```
public class SequenceNode : BehaviorNode // 시퀀스노드 : 모든 자식 노드들을 순차적으로 참조 6개
{
    private int current = 0;
    참조 0개
    public override Status Evaluate()//상태 평가
    {
        if(current >= children.Count)// current는 자식노드가 모두 성공했을 때 자식 노드
        {
            current = 0;//다시 0으로 초기화
            return Status.Success;//current값으로 성공 여부를 구별한다.
        }
        Status childStatus = children[current].Evaluate();//각 자식 노드가 성공했는지
        switch(childStatus)
        {
            case Status.Running :
                return Status.Running;//실행 중일 경우
            case Status.Success :
                current++;
                return current >=children.Count ? Status.Success : Status.Running;
            case Status.Failure :
                current=0;
                return Status.Failure;//실패일 경우
            default : return Status.Success;// 플레이어가 감지되지 않았거나 특정 행동들
```

문제 해결 과정

3

해결 방법 고안

- 플레이어 ↔ 에너미 간 거리를 체크하는 `CheckPlayerDistanceNode`의 `Evaluate()`가 매 프레임 재평가되어야 거리에 따른 Success / Failure가 반환되어 상태 전환 수행이 가능합니다.
- 이를 위해 IDLE 상태일 때 Success를 반환하여 IDLE을 유지하고, 플레이어가 근접하면 Failure를 반환하여 combatSelector로 넘어가 공격 / 추격을 수행하도록 하여 문제를 해결했습니다.

[CheckPlayerDistanceNode]

```
public override Status Evaluate() // 평가함수 오버라이드. IDLE상태로 갈지, 공격 상태로 넘어갈지 체크한다.  
{  
    float currentDistance = Vector3.Distance(shaman.transform.position, shaman.player.position); // 두 캐릭터 간 거리 계산  
  
    Debug.Log($"플레이어와의 거리 : {currentDistance}, 탐지 거리 : {distance}");  
  
    bool inRange = currentDistance <= distance; // 현재 두 캐릭터 간 거리가 탐지 거리 이하일 때 (간격에 들어왔을 때) true  
  
    return (invertCheck ? !inRange : inRange) ? Status.Success : Status.Failure;  
  
    // 플레이어가 탐지 범위 밖이면 (!InRange = true) -> Success  
    // 플레이어가 탐지 범위 안이면 (!InRange = false) -> Failure  
}
```

플레이어 근접 시 `currentDistance <= distance`를 만족 → `isRange = true`

`invertCheck = true`가 전달됨 → 플레이어가 탐지 범위 내에 존재하므로 `Status.Failure`를 반환. → `CombatSelector`로 이동합니다.

```
public class IdleActionNode : BehaviorNode  
{  
  
    // idle 애니메이션을 실행하는 노드  
    private ShamanController shaman;  
  
    public IdleActionNode(ShamanController shaman)  
    {  
        this.shaman = shaman;  
    }  
  
    public override Status Evaluate()  
    {  
        shaman.animator.SetTrigger("IDLE");  
        return Status.Running;  
    }  
}  
  
  
  
public override Status Evaluate()  
{  
    shaman.animator.SetTrigger("IDLE");  
    return Status.Success;  
} // 12.22 IDLE상태 지속 및 시퀀스 평가 중단 오류 해결
```

문제 해결 과정

1

문제 상황 인식

플레이어 추격(Chase)에 실패한 후 IDLE 상태가 무한 반복되는 문제

2

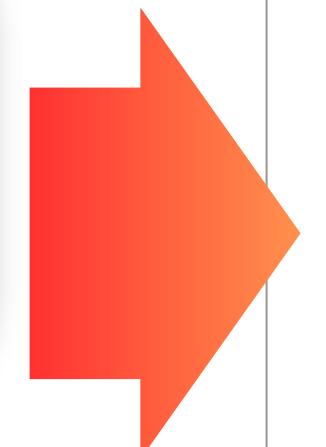
원인 파악

- 플레이어 추격 가능 범위 이탈 시, 에너미는 랜덤 위치로 이동 후 플레이어를 다시 기다려야 합니다.
- 기존 코드에서는 NavMesh.SetDestination()으로 랜덤 위치로 이동하며 Failure를 반환하기 때문에 부모 노드가 추적 실패로 인식하고 IDLE 상태로 전환됩니다.

```
public override Status Evaluate() //플레이어와의 거리 확인
{
    float distanceToPlayer = Vector3.Distance(shaman.transform.position, shaman.player.position);

    if(distanceToPlayer > respawnDistance) //너무 멀어지면 랜덤 위치로 이동하여 대기하도록 한다.
    {
        NavMeshHit hit;
        Vector3 randomDirection = Random.insideUnitSphere * respawnDistance; //네비메시 위의 원형 랜덤 구간 * 거리
        randomDirection+=shaman.player.position; //Navmesh위의 새로운 위치로 이동.

        if(NavMesh.SamplePosition(randomDirection, out hit, respawnDistance, NavMesh.AllAreas))
        {
            shaman.agent.SetDestination(hit.position); //랜덤 위치가 존재할 시 해당 장소로 이하여 대기.
            hasRoared = false; //포효 상태 초기화
            return Status.Failure;
        }
    }
}
```



```
public override Status Evaluate()
{
    //플레이어와의 거리 확인
    float distanceToPlayer = Vector3.Distance(shaman.transform.position, shaman.player.position);

    if(distanceToPlayer > respawnDistance) //너무 멀어지면 리스폰되거나, 그 자리에서 IDLE상태가 된다.
    {
        int maxAttempts = 10; //랜덤 위치를 찾기 위한 시도 횟수에 제한을 둔다.
        int attempts = 0;
        NavMeshHit hit;

        while(attempts < maxAttempts) //시도 제한 횟수까지 반복하며 리스폰 가능 위치를 찾는다.
        {
            Vector3 randomDirection = randomDirection.insideUnitSphere * respawnDistance;
            randomDirection += shaman.player.position;

            //SamplePosition 메서드를 사용하여 최대 respawnDistance 까지 탐색하며 랜덤 위치를 찾는다.
            if(NavMesh.SamplePosition(randomDirection, out hit, respawnDistance, NavMesh.AllAreas))
            {
                shaman.transform.position = hit.position; //에너미 위치를 랜덤 위치로 리스폰.
                shaman.agent.ResetPath(); //기존 경로 초기화
                hasRoared = false; //포효 상태 초기화

                return Status.Success; //리스폰 후 Idle 상태로 전환
            }
            attempts++;
        }
        shaman.agent.ResetPath(); //이동 중지
        hasRoared = false; //포효 상태 초기화
        return Status.Success; //유효한 위치를 찾지 못한 경우, 현재 위치에서 Idle 상태 유지
    }
    shaman.agent.SetDestination(shaman.player.position); //플레이어 추적
}
```

3

해결 방법 고안

- 생성된 랜덤 위치를 Position값에 대입하여 에너미를 리스폰(ReSpawn)시킨 후 IDLE상태로 전환하면 이후 플레이어 재 접근 시 정상적으로 Action이 수행됩니다.
- 이를 위해 Failure 대신 Success를 반환하여야 SequenceNode에서 올바른 상태 전환이 가능합니다.
- 맵의 구조를 고려하여 위치 탐색에 실패할 경우를 대비합니다. 탐색 횟수 제한을 두어 유효 위치를 찾지 못하면, 현재 위치에서 IDLE상태로 전환될 수 있도록 ResetPath() 호출 후 Success를 반환합니다.

Perlin Noise 알고리즘을 활용한 피격 시 카메라 진동 효과 구현

카메라의 급격한 위치 변화 없이 점진적인 흔들림이 유지되는 자연스러운 진동을 구현하였습니다.

```
private void Start()
{
    //펄린노이즈 적용을 위해 랜덤한 시작점을 설정
    noiseOffsetX = Random.Range(0f, 1000f);
    noiseOffsetY = Random.Range(0f, 1000f);
}

public void StartShake()// 공격을 받으면 카메라 흔들림을 시작. DecreaseSanity()가 호출될 때 호출.
{
    StopAllCoroutines(); //이전 흔들림이 있다면 중지함.
    StartCoroutine(ShakeCoroutine());
}
```

참조 1개
private IEnumerator ShakeCoroutine()//펄린 노이즈를 발생시키는 코루틴.

```
{
    float elapsed = 0.0f; //진동 경과 시간
    while(elapsed < shakeDuration)
    {
        elapsed+=Time.deltaTime;
        float percentComplete = elapsed / shakeDuration; //흔들림이 완료되는 시간
        float damper = 1.0f - Mathf.Clamp(percentComplete, 0.0f, 1.0f); //흔들림이 자연스럽게 감소하도록

        //펄린 노이즈를 사용한 흔들림 적용. elapsed*10을 사용하여 노이즈의 속도를 조절한다.
        float offsetX = Mathf.PerlinNoise(noiseOffsetX + elapsed*10f, 0f );
        float offsetY = Mathf.PerlinNoise(0f, noiseOffsetY + elapsed * 10f);

        //1-에서 1 사이의 값으로 변환 후 흔들린 위치를 카메라 로컬포지션에 적용
        offsetX = (offsetX * 2.0f - 1.0f) * shakeAmount * damper;
        offsetY = (offsetY * 2.0f - 1.0f) * shakeAmount * damper;
        cam.transform.localPosition = originalPos + new Vector3(offsetX, offsetY, 0);

        yield return null;
    }
    cam.transform.localPosition = originalPos; //카메라 원래 위치로 이동.
}
```

[의도]

- 플레이어가 피격 당할 때 카메라를 흔들어 **충격을 직관적으로 전달하는 효과**를 주고자 하였습니다.
- 단순한 카메라 진동은 Random 난수 값을 사용하지만, Perlin Noise를 사용하면 $0.3 \rightarrow 0.35 \rightarrow 0.4 \rightarrow 0.5 \dots$ 처럼 **특정 위치에서 연속된 랜덤값**을 얻을 수 있습니다.

[동작 원리]

- X, Y좌표의 난수 시작점을 설정합니다. 만약 5.0f 이면 난수는 5.0f → 5.5f...으로 생성됩니다.
- **전체 지속 시간 대비 진행 비율**을 elapsed / shakeDuration으로 구한 후 **damper** 변수에 적용합니다. 처음에 강했던 진동이 시간에 따라 감소합니다.
- **Mathf.PerlinNoise()**를 사용하여 시작점부터 연속적인 난수를 생성합니다. 이를 (-1, 1) 범위로 변환 후 shakeAmount와 damper를 곱하여 적용합니다.

[결과]

- 랜덤한 값이 아닌 **부드러운 곡선을 따라 진동하는 현실적인 충격 효과**를 구현하였습니다.
- elapsed * 10f 값을 조절하여 진동 속도 변경이 가능하며, 흔들림은 시간에 따라 자연스럽게 감소하므로 3D 어지러움과 같은 현상을 예방할 수 있습니다.

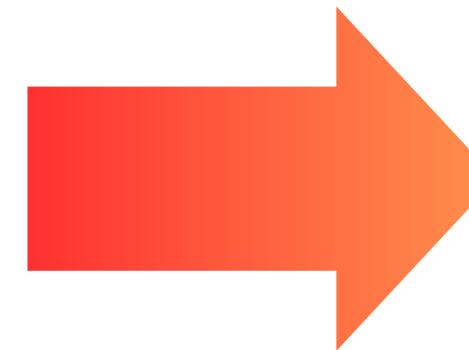
[문제점 개선]



Perlin Noise 알고리즘을 활용한 피격 시 카메라 진동 효과 구현

불필요한 코루틴 종료 문제점 개선

```
public void StartShake()// 공격을 받으면 카메라 흔들림을 시작.  
{  
    StopAllCoroutines(); //이전 흔들림이 있다면 중지함.  
    StartCoroutine(ShakeCoroutine());  
}
```



```
public void StartShake()// 공격을 받으면 카메라 흔들림을 시작.  
{  
    ShakeCoroutine prevCoroutine = ShakeCoroutine();  
  
    StopCoroutine(prevCoroutine); //이전 흔들림이 있다면 중지함.  
  
    StartCoroutine(ShakeCoroutine());  
}
```

- 기존 코드에서 `StopAllCoroutines()`으로 진동을 초기화했습니다.
- 하지만 이는 현재 실행 중인 다른 코루틴도 중지할 위험이 있는데, `Coroutine` 변수에 할당한 `ShakeCoroutine()`만 중지하는 방식으로 개선하였습니다.

[Perlin Noise 진동 효과 적용 결과]  [YouTube 링크](#)



던전 키우기

#2D #픽셀 아트 #방치형 RPG #모바일 #Google Play 출시작

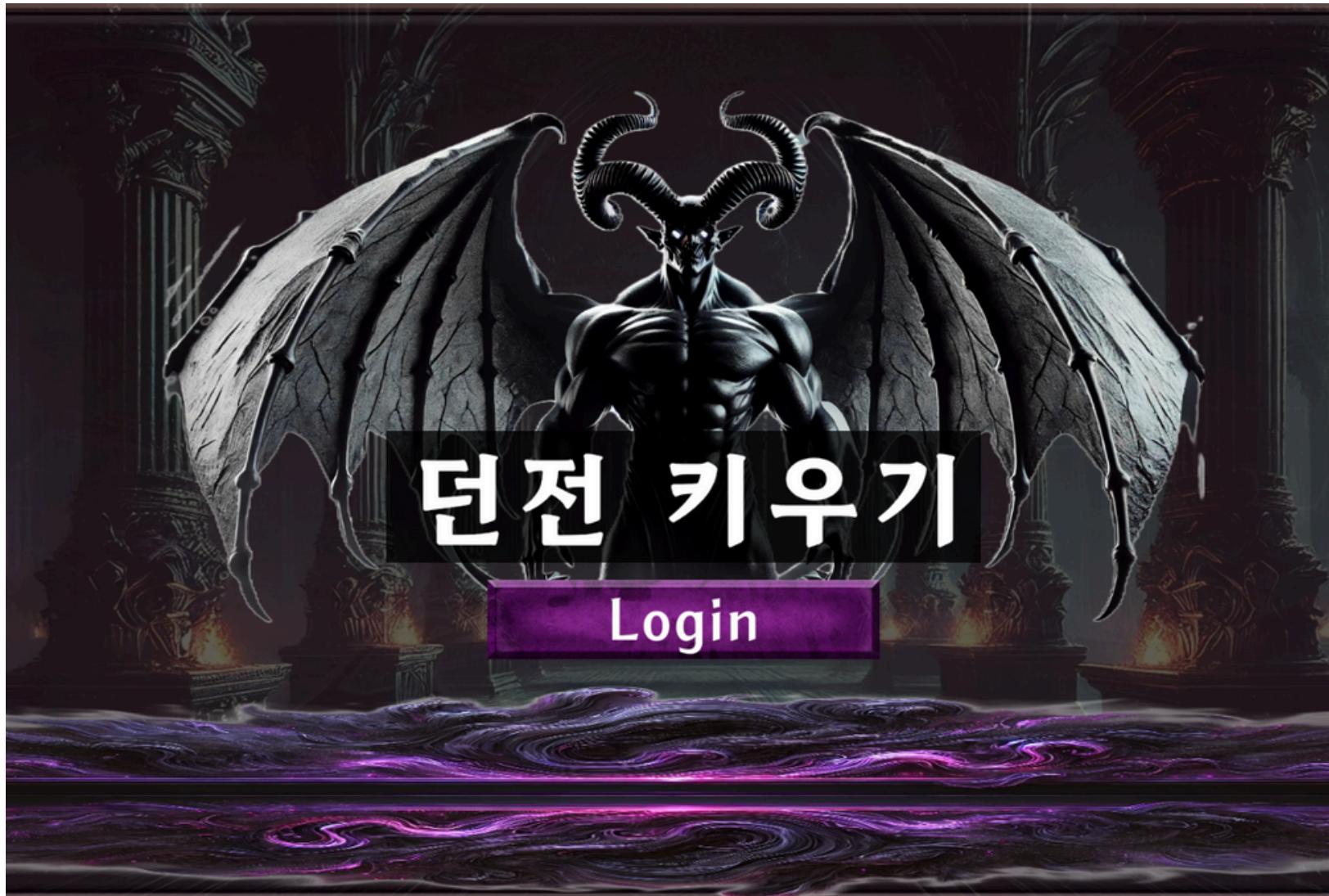
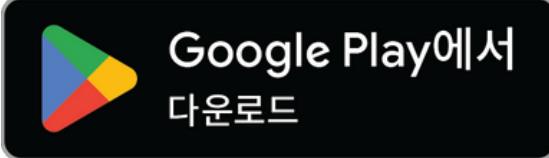
Unity

C#

Git



[GitHub](#)



담당 업무

[클라이언트 개발]

- 데이터 기반 정보 관리 시스템 개발
 - Scriptable Object를 활용한 인 게임 데이터 관리 구현
 - 디자인 패턴 기반 퀘스트 시스템 구현
- 2D 픽셀 아트 컨셉의 전투 시스템 개발
 - Tilemap을 사용한 전투 환경 구현
 - Tilemap 환경 내 유닛 배치 알고리즘 구현
 - 방치형 자동 전투 시스템 구현
- UGUI를 사용한 UI 개발
 - 퀘스트 / 인벤토리 / 대장간 등 게임 내 전체 UI 구현
- 서드파티 SDK 활용
 - 뒤끝(The Backend) SDK를 활용한 비속어 필터링 기능 구현
 - SPUM을 활용한 2D 픽셀 캐릭터 제작

[프로젝트 관리]

- Google Play 출시 전담

Scriptable Object를 활용한 인 게임 데이터 관리 구현

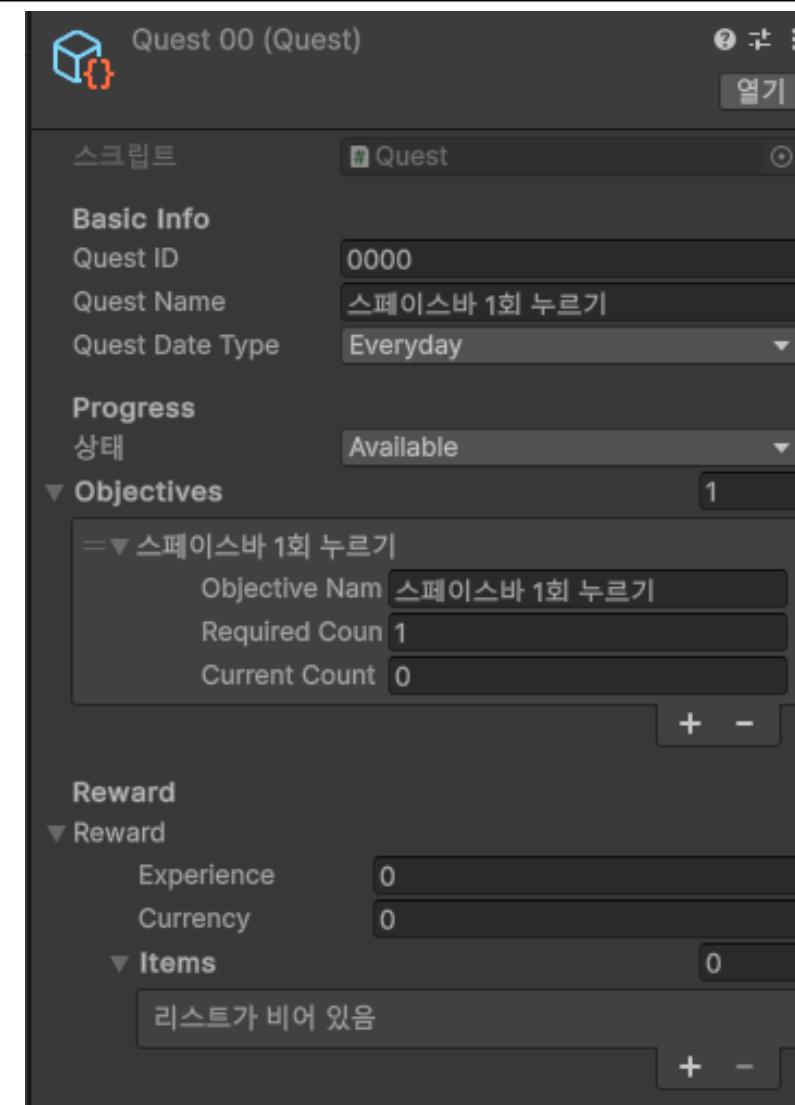
(1) 퀘스트 데이터 관리 체계를 확립하였습니다.

```
[CreateAssetMenu(menuName = "Quest/New Quest")]
참조 18개
public class Quest : ScriptableObject
{
    //퀘스트 데이터를 저장하기 위한 스크립터를 오브젝트

    [Header("Basic Info")]
    참조 0개
    public string questID; //퀘스트의 고유한 id
    참조 4개
    public string questName; //퀘스트 이름
    참조 0개
    public QuestDate questDateType; //일일, 주간 퀘스트 구분

    [Header("Progress")]
    참조 11개
    public QuestStatus status; //퀘스트 진행 상태
    참조 7개
    public QuestObjective[] objectives; //퀘스트 성공 조건(목표)

    [Header("Reward")]
    참조 0개
    public QuestReward reward; //퀘스트 보상
}
```



[의도]

- 정해진 목표, 보상, 조건을 갖는 퀘스트를 SO로 관리하는 직관적인 구조를 지향하였습니다.
- Scene 전환 후에도 퀘스트 데이터를 유지하고자 하였습니다.
- 불변 데이터인 퀘스트는 SO, 가변 데이터인 진행 상태는 PlayerPrefs로 관리하여 개발 편의성을 높이고자 하였습니다.

[결과]

- MVC 패턴 기반 [퀘스트 데이터 - 로직 - 인벤토리 UI] 구조 확립
- QuestManager를 통해 외부 시스템과의 간단한 상호작용이 강점입니다.

디자인 패턴 기반 쿼스트 시스템 구현

(2) Facade 패턴을 적용한 쿼스트 시스템 진입점을 설계하였습니다.

```
public class QuestManager : MonoBehaviour
{
    참조 1개
    public static void AcceptQuest(Quest quest) => QuestStatusManager.Instance.AcceptQuest(quest); //퀘스트 수락 메서드 --> 퀘스트 진행도 상태가 Available일 경우 수락. 중복 수락x
    참조 1개
    public static void UpdateQuestProgress(Quest quest, int objectiveIndex, int amount) => QuestStatusManager.Instance.UpdateObjectiveProgress(quest, objectiveIndex, amount); //퀘스트 진행도 업데이트 메서드
    //quest : 대상 퀘스트, objectiveIndex : 업데이트할 목표 인덱스, amount : 증가 수량 또는 횟수
    참조 1개
    public static void CompleteQuest(Quest quest) => QuestStatusManager.Instance.CompleteQuest(quest);
    참조 0개
    public static string GetQuestStatus(Quest quest) => QuestStatusManager.Instance.GetQuestStatus(quest); //외부에서 현재 퀘스트 상태에 접근할 때 사용하는 메서드.
    /* ...
}
```

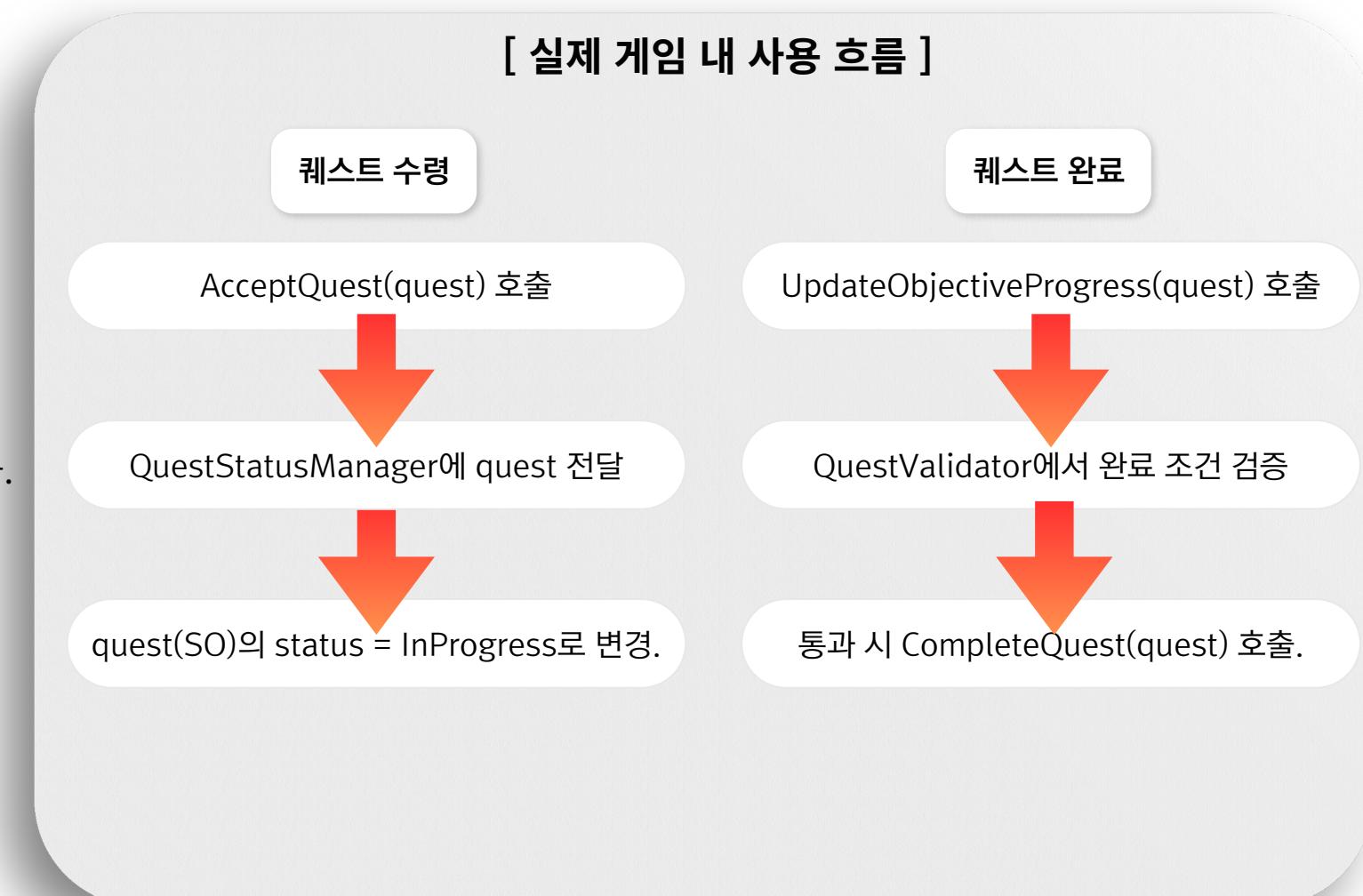
[의도]

- 외부에서 퀘스트 관련 작업 호출 시, 내부 클래스를 알 필요 없는 간소화된 인터페이스를 설계하고자 하였습니다.
- 단일 책임 원칙을 철저히 지키는 중계 클래스를 통해 개발의 편의성을 확보하고자 하였습니다.

[결과]

- 클래스 별 명확한 책임 분리 구현
 - 용이한 확장성
 - QuestManager : 시스템 호출 중계
 - 새로운 기능 추가 시 QuestManager 수정 없이 하위 클래스만 확장합니다.
 - QuestStatusManager : 상태 변경 로직 선언
 - ex) `public static void AcceptQuest(Quest quest)`
 - {
 - `QuestStateManager.Instance.AcceptQuest(quest);`
 - `SaveSystem.SaveQuestProgress(quest); // 저장 기능 확장`
 - }
 - QuestValidator : 완료 조건 검증
 - RewardHandler : 보상 처리

[실제 게임 내 사용 흐름]



Scriptable Object를 활용한 인 게임 데이터 관리 구현

(3) MVC 패턴을 적용한 퀘스트 데이터 관리 아키텍처를 구현하였습니다.

• 퀘스트 수락

→ 해당 퀘스트를 “진행 중”으로 변경 후 활성화 리스트에 추가합니다.

```
private List<Quest> activeQuests = new(); //Quest타입 리스트변수 -> 현재 활성화된 퀘스트 인스턴스를 생성.  
public static event Action<Quest> OnQuestUpdated; //퀘스트 진행도 변경 시 UI에 실시간 반영을 위한 이벤트.  
  
public void AcceptQuest(Quest quest) //퀘스트 수락 메서드  
{  
    if(quest.status != QuestStatus.Available)  
        return; //현재 수락 가능 퀘스트가 아닐 경우 함수 종료.  
  
    quest.status = QuestStatus.InProgress; //수락 가능 -> 진행 중으로 상태 변경  
    activeQuests.Add(quest); //활성화된 퀘스트 리스트에 추가.  
    Debug.Log($"Quest Accepted : {quest.questName}"); //퀘스트 명을 로그로 표시.  
}
```

• 완료 조건 검증

→ 플레이어의 달성 현황과 실제 퀘스트 데이터 상 달성 조건의 일치 여부를 검사합니다.

```
public static class QuestValidator  
{  
    //퀘스트 검증 클래스. 완료 상태를 검증하고, 보상 지급 함수를 호출한다. QuestStatusManager 클래스의 UpdateObjectiveProgress에서 호출됨.  
    참조 0개  
    public static void CheckCompletion(Quest quest) //퀘스트 종료 전 실제 퀘스트가 완료되었는지 체크.  
    {  
        if(quest.status != QuestStatus.InProgress)  
            return; //퀘스트가 현재 진행 중인 상태가 아닐 경우 종료.  
  
        foreach(var objective in quest.objectives) //퀘스트 달성을 위해 요구되는 목표 수량, 횟수가 현재 수량, 횟수 이상일 경우(즉, 아직 달성 조건 불만족)  
        {  
            if(objective.currentCount < objective.requiredCount) // ex ) 현재 공격 횟수 40회 < 목표 달성 횟수 50회  
                return;  
        }  
        QuestStatusManager.Instance.CompleteQuest(quest); //퀘스트 진행 중 여부 검증 -> 진행 중이 아니면 퀘스트를 완료 상태로 변경  
        RewardHandler.GiveReward(quest.reward); //보상 지급 함수를 호출.  
    }  
}
```

• 퀘스트 완료

→ 활성화된 퀘스트의 완료 조건 충족 시 “완료” 상태로 변경 후 비활성화 합니다.

```
public void CompleteQuest(Quest quest) //퀘스트 완료  
{  
    if(quest.status != QuestStatus.InProgress)  
        return; //퀘스트가 현재 진행 중이 아니라면 함수 종료. 현재 진행 중인 퀘스트만 완료될 수 있다.  
  
    quest.status = QuestStatus.Completed; //퀘스트를 완료 상태로 바꾼다.  
    activeQuests.Remove(quest); //활성화 퀘스트 목록에서 제거.  
    Debug.Log($"Quest Completed : {quest.questName}");  
}
```

• 퀘스트 진행도 갱신

→ 퀘스트 요구 조건 충족 시 호출하여 진행도를 갱신하고 완료 조건을 검증합니다.

```
public void UpdateObjectiveProgress(Quest quest, int objectiveIndex, int amount) //퀘스트 진행상황 업데이트 메서드.  
{  
    if(quest.status != QuestStatus.InProgress)  
        return;  
  
    quest.objectives[objectiveIndex].currentCount += amount; //퀘스트 달성을 위한 최소 횟수, 수량 등을 amount값을 더해 업데이트.  
    OnQuestUpdated?.Invoke(quest); //진행도 업데이트 시 이벤트 발생  
  
    QuestValidator.CheckCompletion(quest); //퀘스트 진행도와 currentCount값이 requiredCount값을 만족 하는지 검증한다. 검증 완료 시 보상
```

Tilemap 환경 내 유닛 배치 알고리즘 구현

게임 시작 시 유닛 스폰 및 자동배치 기능을 구현하였습니다.

```
private void FindSpawnPosition(), //병사 프리팹을 타일맵 위 위치에 보유 숫자만큼 스폰한다.  
{  
    //BoundsInt는 주로 타일맵 내의 유효한 cell 영역을 나타내는데 사용. 각 cell은 하나의 tile을 나타낸다.  
    BoundsInt bounds = soldierTilemapLayer.cellBounds;  
    foreach(Vector3Int position in bounds.allPositionsWithin)  
    {  
        if(soldierTilemapLayer.HasTile(position)) //타일맵 셀에 존재하는 모든 위치를 꺼내어 반복하며 타일이 존재하는 위치를 저장  
        {  
            soldierSpawnPositions.Enqueue(position); //병사 소환용 레이어에서 타일이 존재하는 위치만 좌표저장 큐에 삽입.  
            //Debug.Log($"soldierSpawnPositions.Count = {soldierSpawnPositions.Count}");  
        }  
    }  
    if(soldierSpawnPositions.Count == 0)  
    {  
        Debug.LogWarning("There is no tile to spawn soldier.");  
    }  
}  
  
private void SpawnSoldier(), //병사의 소환 가능 여부를 체크하고 소환 가능한 타일에 병사 프리팹을 Instantiate()하는 함수  
{  
    if(spawnedCount >= maxAmount)  
    {  
        Debug.LogWarning("maxAmount of soldier is spawned.");  
        return;  
    }  
    /*  
     * 캐릭터가 스폰될 타일은 FindSpawnPosition()에서 큐에 넣어진 타일이며, 순서대로 유닛이 배치된다.  
     * Dequeue로 첫 좌표부터 꺼내고 나면 캐릭터가 위치한 타일을 제거하여 사용 불가하게 함. (같은 타일에 유닛이 중복 소환되는 현상 방지)  
     */  
    if(soldierSpawnPositions.Count > 0)  
    {  
        Vector3Int spawnTile = soldierSpawnPositions.Dequeue();  
        Vector3 worldPosition = soldierTilemapLayer.GetCellCenterWorld(spawnTile); //해당 타일 중심 위치를 가져와서 그곳에 캐릭터를 배치.  
        GameObject newSoldier = Instantiate(mySoldiers[0], worldPosition, rotation, prefabParent); //병사 유닛 리스트의 첫번째 요소를 스폰.  
        spawnedSoldiers.Add(newSoldier);  
        Debug.Log($"soldierSpawnPositions.Count = {soldierSpawnPositions.Count}");  
        spawnedCount++;  
    }  
    else  
    {  
        Debug.LogWarning("There is no tile to spawn soldier.");  
    }  
}
```



[의도]

- 던전 첫 진입 시 보유한 유닛 객체가 자동으로 Tilemap Layer에 배치되도록 의도하였습니다.

[동작 원리]

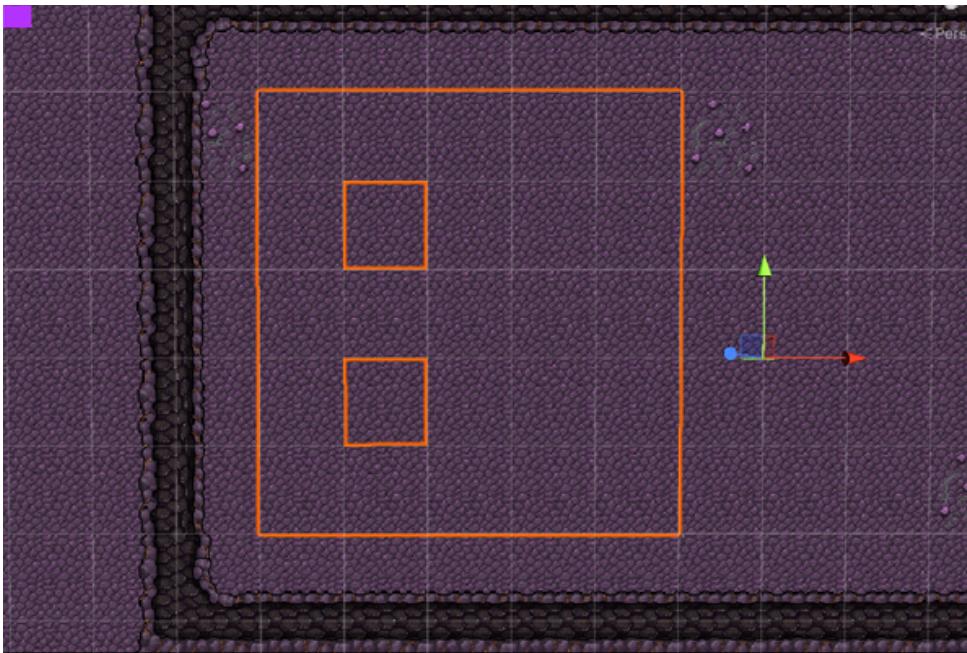
- Scene 활성화 시 FindSpawnPosition()을 호출하여 유닛 배치 가능한 타일 위치를 탐색하여 Queue에 저장합니다.
- SpawnSoldier()에서 타일 좌표를 차례로 Dequeue하여 해당 타일의 월드 좌표를 GetCellCenterWorld() 메서드로 계산 후 유닛을 인스턴스화 합니다.
- 좌표는 Dequeue로 꺼내진 후 즉시 제거되므로 같은 타일 위 유닛의 중복 소환을 방지합니다.

[결과]

- Scene 활성화 시 최대 10개의 병사 유닛이 스폰됩니다.
- 같은 위치에 유닛이 중복 배치되지 않습니다.



원인 분석 및 해결

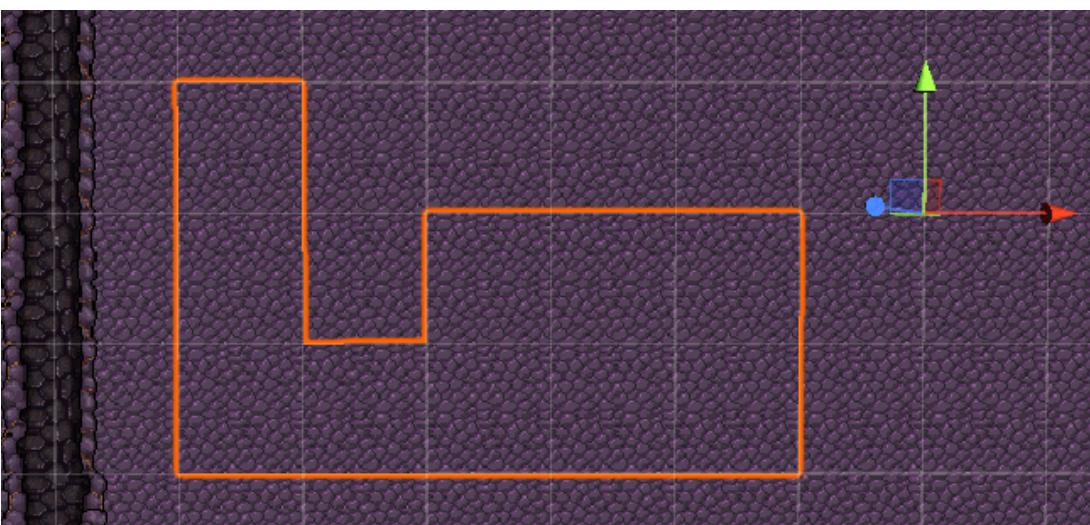


- 병사 유닛 배치를 위해 할당된 타일 수는 **23칸**입니다.
- 자동 배치 시, 스폰 좌표 탐색 과정에서 23칸의 좌표가 큐에 추가되며, **배치 메서드**가 실행될 때 배치 가능한 **10개**의 좌표만 Dequeue됩니다.
이때 큐에는 **13개의 좌표가 남아** 있습니다.
- [자동 배치 ↔ 초기화]를 반복할 때마다 13개 좌표가 채워진 큐에 **다시 23개 좌표가 Enqueue**되고, **10개씩 Dequeue**되는 현상이 반복됩니다.
- 결과적으로 **초기화**가 실행될 때마다 큐 공간이 **13개씩 증가**하게 됩니다.

자동 배치	초기화
10칸 Enqueue후 10칸 Dequeue	23칸 Enqueue후 10칸 Dequeue

BoundsInt.allPositionsWithin의 이해를 통한 해결

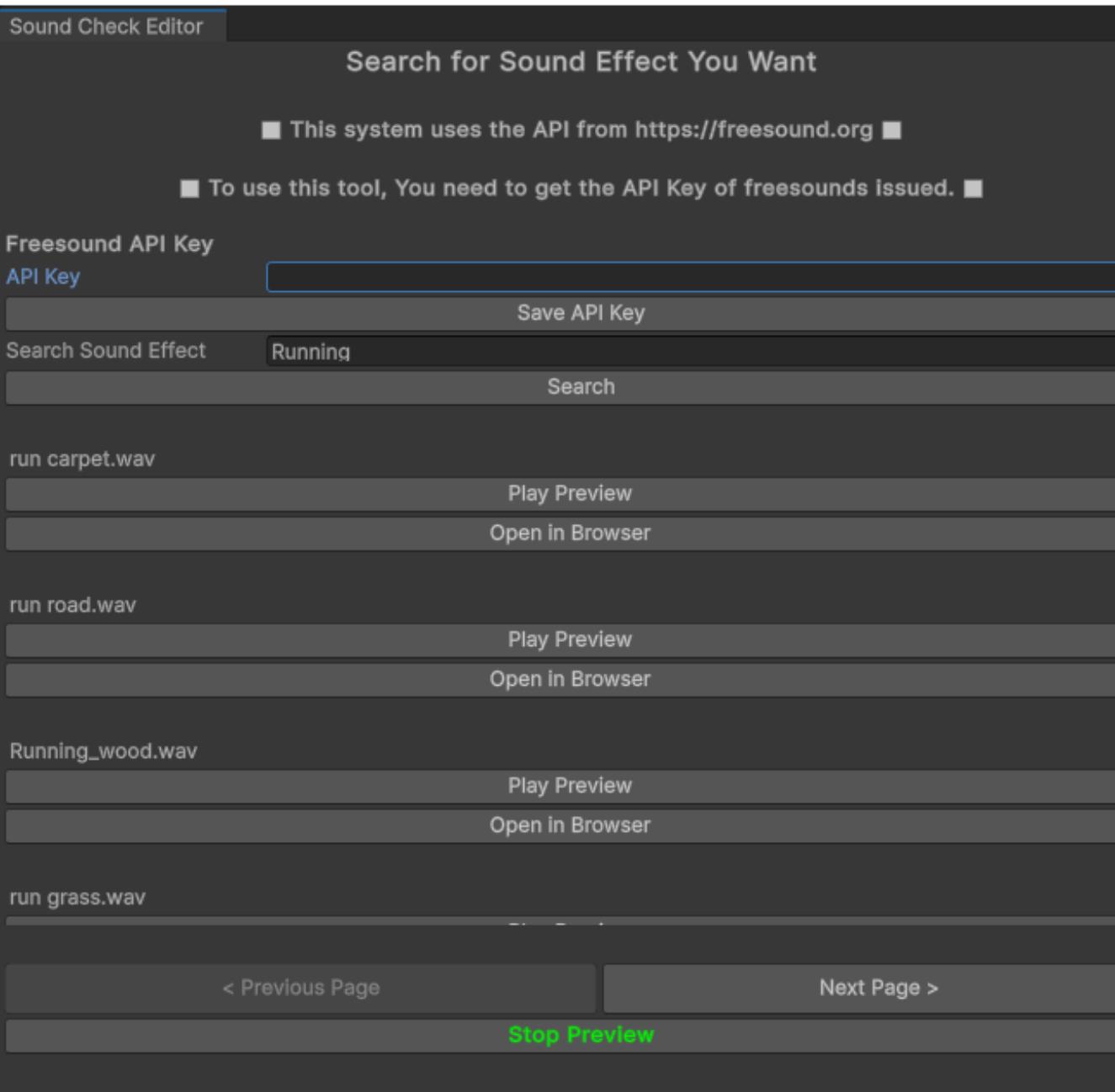
- Tilemap의 Cell 영역을 나타내는 BoundsInt의 **allPositionWithin** 메서드는 Tilemap Layer의 좌 하단에서 시작하여 우 상단까지 Vector3Int 좌표를 탐색합니다.
- 따라서 초기화 스폰 지점은 항상 좌 하단부터 10칸이 됩니다.
- 이 점에 착안하여, **좌 하단부터 10칸의 초기화용 Layer를 생성**하는 것으로 문제를 해결하였습니다.



```
private void FindSoldiersPosition(Tilemap tilemap)//병사 유닛들의 스폰 장소를 찾는다.
{
    //BoundsInt는 주로 타일맵 내의 유효한 cell 영역을 나타내는 데 사용. 각 cell은 하나의 tile을 나타낸다.
    BoundsInt bounds = tilemap.cellBounds;
    foreach(Vector3Int pos in bounds.allPositionsWithin)//자동배치용 병사타일에서 배치 가능한 위치를 찾는다.
    {
        if(tilemap.HasTile(pos))
        {
            availableTilesForSoldiers.Enqueue(pos); //타일목록 큐에 삽입.
            Debug.Log($"availableTilesForSoldiers.Count = {availableTilesForSoldiers.Count}");
        }
    }
    if(availableTilesForSoldiers.Count == 0)
    {
        Debug.LogWarning("No Available Tiles found in soldierPlaceLayer");
    }
}
```

→ 기존 Layer를 10칸으로 줄이지 않고 새 Layer를 생성한 이유?

- 총 25칸(병사 23칸 + 영웅 1칸 + 캐릭터 1칸)의 좌표가 모두 저장된 Layer가 존재해야 드래그를 통한 수동 배치가 가능하기 때문입니다.
- 만약 기존 23칸 Layer를 10칸으로 변경할 경우, 드래그 수동 배치가 가능한 Tile은 12칸으로 제한됩니다.



SoundCheckEditor

Unity 기반 Freesound API 통합 사운드 검색 도구

Unity

C#

HTTP

Freesound API



[GitHub](#)

MADE WITH



Unity® Asset Store

담당 업무

[클라이언트 개발]

- Unity 에디터 확장 기능 개발
 - 검색어 입력, 검색 결과 목록, 미리듣기 버튼, 페이지네이션 구현
- 외부 API 통합 및 HTTP 통신
 - API 요청 및 응답 처리 기능 구현
 - 사운드 미리듣기를 위한 오디오 스트리밍 기능 구현
- 사용자 경험 개선
 - HTTP 요청 진행도 표시기 구현
 - HttpRequestException 및 JsonException 에러 핸들링

[프로젝트 관리]

- Unity Asset Store 퍼블리싱

개발 목적

- Unity의 EditorWindow 시스템을 활용하여 **개발자 툴 제작 역량**을 쌓고자 하였습니다.
- 엔진 바깥에서 사운드 리소스를 탐색하며 **불편함을 겪었던 경험에서 아이디어**를 얻었습니다.
- 게임 개발 실무에 바로 사용할 수 있는 효율적인 기능 구현을 목표로 하였습니다.

API 요청 및 응답 처리 기능 구현

(1) 사용자 검색어 인코딩 및 요청 전송 기능을 구현하였습니다.

```
searchQuery = EditorGUILayout.TextField("Search Sound Effect", searchQuery); // 검색어 입력

private async void SearchSound(string query)// Freesound의 api키와 검색 쿼리를 사용하여 api요청을 보내 사운드를 검색. 입력한 데이터에 따라 동적으로 URL 생성,호출
{
    CheckApiKeyValidation(); //API KEY 유효성 검사 수행.
    if(string.IsNullOrEmpty(query))//쿼리가 비어있다면
    {
        EditorUtility.DisplayDialog("Empty Query", "Please enter Query in blank.", "OK");
        return;
    }
    string encodedQuery = HttpUtility.UrlEncode(query);
    /* 쿼리는 URL형태로 전송되므로, 특수문자나 공백이 포함될 경우를 대비한 URL인코딩이 필요.
     * C#의 System.Web에 있는 HttpUtility.UrlEncode를 사용. 이를 통해 서버와의 통신 오류, 검색 결과 누락, 보안 문제 등 방지.
     * 저작권이 자유로운 cc0라이선스를 필터링, api응답에 프리뷰 url을 포함하여 요청하기 위해 프리뷰 파라미터 추가*/
    string url = $"https://freesound.org/api/v2/search/text/?query={encodedQuery}&filter=license:(\"Creative Commons 0\")&fields=id,name,previews,username";

    var request = new HttpRequestMessage(HttpMethod.Get, url); // HttpRequestMessage 객체 생성 및 인증 헤더 설정
    request.Headers.Add("Authorization", $"Token {apiKey}");
    await FetchSoundData(request); // 주어진 url로 freesound 데이터를 검색하는 메서드. 페이지 이동 시 사용
}
```

[동작 원리]

- OnGUI()에서 API에 HTTP 요청을 보낼 쿼리를 입력 받습니다.
- 쿼리에 특수문자나 공백이 포함되어 있을 경우 오류가 발생할 수 있으므로 `HttpUtility.UrlEncode`로 인코딩합니다. 예를 들어, “`Sound Effect!`”는 “`Sound%20Effect%21`”로 변환됩니다.
- 상업적 이용에 제한이 없는 CC0 라이선스 사운드만 필터링 하여 사운드의 ID, 이름, 미리듣기 소스, 업로더 명을 쿼리와 함께 URL에 삽입합니다.
- `HttpRequestMessage` 객체를 생성하고 인증 정보를 헤더에 추가한 후, 비동기 요청을 위해 `FetchSoundData()`에 전달합니다.

API 요청 및 응답 처리 기능 구현

(2) HTTP 요청 전송 및 응답 처리, 에러 핸들링 기능을 구현하였습니다.

```
private void HandleApiError(HttpResponseMessage response)//api키 메러 핸들링을 위한 메서드
{
    string errorMessage = $"API Error : {response.ReasonPhrase}";

    switch (response.StatusCode)//인증오류 시
    {
        case System.Net.HttpStatusCode.Unauthorized:
            errorMessage = "Unauthorized: Your API Key might be invalid or expired.";
            break;
        case System.Net.HttpStatusCode.Forbidden:
            errorMessage = "Forbidden: You do not have permission to access this resource.";
            break;
    }

    Debug.LogError(errorMessage);
    EditorUtility.DisplayDialog("API Error", errorMessage, "OK");
}

private async Task FetchSoundData(HttpRequestMessage request) //주어진 URL로 freesound 데이터를 비동기적으로 가져오는 메서드
{
    try
    {
        isSearching = true;//검색 시작. HTTP 요청 전후로 진행상태를 업데이트.
        progress = 0.0f;
        progressMessage = "Sendig Request...";

        HttpResponseMessage response = await client.SendAsync(request); //비동기적으로 HTTP GET 요청을 보낸 후 응답을 받는다. 생성한 URL을 사용
        progress = 0.5f;
        progressMessage = "Receiving data...";

        if (response.IsSuccessStatusCode) // 요청이 성공했는지 확인
        {
            string jsonResponse = await response.Content.ReadAsStringAsync(); //응답의 JSON 문자열을 읽는다.
            progress = 0.8f;
            progressMessage = "Parsing data...";

            soundResults = ParseSoundResults(jsonResponse); //JSON 문자열을 파싱하여 검색 결과 리스트를 업데이트
            progress = 1f;
            progressMessage = "Complete!";
        }
        else
        {
            HandleApiError(response);
        }
    }
    catch (HttpRequestException e)//요청 실패 시
    {
        Debug.LogError($"Error fetching sound data: {e.Message}");
        EditorUtility.DisplayDialog("API Request Error", $"Error fetching sound data: {e.Message}", "OK");
    }
    finally
    {
        isSearching = false; // 검색 종료
        progress = 0f;
        progressMessage = "";
    }
}
```

[의도]

- API에 HTTP 요청을 보내고 응답 데이터를 받아와 검색 결과를 처리합니다.
- 비동기적으로 동작하여 에디터 중지 없이 작업을 수행합니다.
- 진행 상태를 실시간으로 업데이트하여 사용자에게 피드백을 제공합니다.

[동작 원리]

- SearchSound()에서 API Key가 포함된 요청을 전달 받아 HttpClient.SendAsync()를 사용하여 HTTP GET 요청을 비동기 전송합니다.
- 응답 성공 시 response.Content.ReadAsStringAsync()를 사용하여 JSON 문자열을 읽고, ParseSoundResults() 함수를 호출하여 검색 결과를 파싱합니다.
- 응답 실패 시 에러 핸들러 함수를 통해 response의 HttpStatusCode를 파악합니다. API Key 오타 혹은 만료 시 Unauthorized, API 접근 권한 문제일 경우 Forbidden으로 감지되어, 이를 사용자에게 안내합니다.

API 요청 및 응답 처리 기능 구현

(3) JSON 데이터 파싱 및 검색 결과 처리 기능을 구현하였습니다.

```
private List<SoundResult> ParseSoundResults(string json)//freesound API 응답을 파싱하여 검색 결과를 반환하는 메서드
{
    if (string.IsNullOrEmpty(json))//api 응답 데이터 유효성 검증 --> json파싱 전에 api 응답이 empty인 경우를 방지
    {
        Debug.LogError("API response is empty.");
        EditorUtility.DisplayDialog("API Error", "The API response is empty.", "OK");
        return new List<SoundResult>();
    }
    try
    {
        //JSON 문자열을 FreesoundSearchResult 객체로 변환.
        var searchResults = JsonConvert.DeserializeObject<FreesoundSearchResult>(json);
        if (searchResults == null || searchResults.results == null || searchResults?.results == null)
        {
            Debug.LogError("No search results found or invalid JSON structure.");// 검색 결과가 없거나 json 구조가 유효하지 않을 경우 출력
            return new List<SoundResult>();//빈 리스트를 반환도록 한다.
        }
        //다음페이지와 이전 페이지의 URL 설정
        nextPageUrl = CleanUrl(searchResults.next);
        prevPageUrl = CleanUrl(searchResults.previous);

        List<SoundResult> results = new List<SoundResult>();

        foreach (var sound in searchResults.results)// 검색 결과 리스트를 순회하여 SoundResult 객체로 변환 후 리스트에 추가
        {
            results.Add(new SoundResult
            {
                id = sound.id,
                name = sound.name,
                license = sound.license,
                username = sound.username,
                previews = sound.previews
            });
        }
        return results;//변환된 검색 결과 리스트 반환
    }
}
```

[의도]

- API로부터 받은 JSON 응답을 Unity에서 사용 가능하도록 **역직렬화** 합니다.
- JSON 데이터에서 사운드 데이터 및 페이지 데이터를 추출합니다.

[동작 원리]

- DeserializeObject()를 사용하여 JSON 문자열을 FreesoundSearchResult 객체로 변환합니다.
- **페이지네이션**을 위해 next, previous 필드 추출 후, 불필요한 부분은 제거합니다.
- 검색 결과 리스트인 searchResults.results를 순회하며 각 사운드 정보를 SoundResult 객체로 변환합니다.

[문제 해결 과정]

- FreesoundSearchResult 클래스는 **네스팅된 사용자 정의타입 리스트**를 포함하고 있기에 JsonUtility 라이브러리 사용 시 오류가 발생합니다.
- 이를 해결하기 위해 .NET Framework 기반의 **NewtonSoft.json 라이브러리**를 사용하여 유연한 역직렬화를 수행했습니다.
- 또한 Assembly Definition 파일에 Newtonsoft.json 참조 추가하여 **사용자가 라이브러리 추가 없이 바로 사용 가능하도록** 조치하였습니다.

```
var searchResults = JsonUtility.FromJson<FreesoundSearchResult>(json);
```

! [01:24:08] No search results found or invalid JSON structure.
UnityEngine.Debug.LogError (object)

using Newtonsoft.Json;
var searchResults = JsonConvert.DeserializeObject<FreesoundSearchResult>(json);

사운드 미리듣기를 위한 오디오 스트리밍 기능 구현

URL으로 다운로드한 오디오를 에디터에서 재생하는 기능을 구현하였습니다.

```
private async void PlayPreview(string previewUrl)//목록에 출력된 사운드 리소스의 미리듣기 기능을 구현하는 메서드.  
{  
    CheckApiKeyValidation(); //API 키 유효성 검사  
    CheckUrlValidation(previewUrl); //URL 유효성 검사  
    try  
    {  
        if(audioSource.isPlaying)//현재 미리듣기 중일 경우 중지.  
        {  
            StopPreview();  
        }  
        //UnityWebRequestMultimedia.GetAudioClip을 사용하여 미리듣기 오디오 클립을 다운로드.  
        using(UnityWebRequest www = UnityWebRequestMultimedia.GetAudioClip(previewUrl, audioType:AudioType.MPEG))  
        {  
            // Freesound API는 미리듣기 요청에도 API 키를 필요로 하므로, 헤더에 API 키를 추가  
            www.SetRequestHeader("Authorization", $"Token {apiKey}");  
  
            var operation = www.SendWebRequest(); //비동기 오디오클립 다운로드.  
            while(!operation.isDone)//다운로드 완료 시 까지 대기.  
            {  
                await Task.Yield(); //비동기 대기.  
            }  
  
            if(www.result == UnityWebRequest.Result.ConnectionError || www.result == UnityWebRequest.Result.ProtocolError)  
            {  
                Debug.LogError($"Error loading audio : {www.error}"); // 다운로드 중 오류 발생 시 메시지 출력.  
                EditorUtility.DisplayDialog("Audio Error", $"Failed to load audio : {www.error}", "OK");  
            }  
            else//다운로드 성공 시  
            {  
                currentAudioClip = DownloadHandlerAudioClip.GetContent(www); //다운로드된 클립을 가져온다.  
  
                if(currentAudioClip != null)//오디오 클립이 정상적으로 로드되었다면  
                {  
                    if(audioSource.isPlaying)// 현재 오디오 재생 중일 경우.  
                    {  
                        audioSource.Stop(); // StopPreview()는 currentAudioClip도 null로 초기화시키므로, 여기서는 호출X  
                    }  
                    audioSource.clip = currentAudioClip;  
                    audioSource.Play();  
                }  
            }  
        }  
    }  
}
```

[의도]

- 미리듣기 기능을 Unity 에디터 내에서 구현하여 사용자가 사운드를 미리듣기할 수 있도록 합니다.

[동작 원리]

- 현재 오디오 재생 여부를 확인하여 중복 재생을 방지합니다.
- UnityWebRequestMultimedia.GetAudioClip()을 사용하여 미리듣기 오디오를 다운로드합니다.
- www.SendWebRequest()를 호출하여 다운로드를 시작하고, await Task.Yield()를 사용하여 다운로드 완료 시 까지 비동기적으로 대기합니다.
- 다운로드 성공 시 DownloadHandlerAudioClip.GetContent(www)를 사용하여 오디오 클립을 가져옵니다.